XAPP1026 (v2.0) June 15, 2009

LightWeight IP (lwIP) Application Examples

Author: Siva Velusamy

Summary

Lightweight IP (lwIP) is an open source TCP/IP networking stack for embedded systems. Xilinx Embedded Development Kit (EDK) provides lwIP software customized to run on Xilinx Embedded systems containing either a PowerPC® or a MicroBlaze™ processor. This application note describes how to utilize the lwIP library to add networking capability to an embedded system. In particular, lwIP is utilized to develop the following applications: echo server, web server, TFTP server and receive and transmit throughput tests.

Included Systems

Included with this application note are reference systems for the Xilinx ML505, ML507 and Spartan®-3AN FPGA Starter Kit boards:

https://secure.xilinx.com/webreg/clickthrough.do?cid=107743.zip

Hardware and Software Requirements

The hardware and software requirements are:

- One of Xilinx ML505, ML507 or Spartan-3AN Starter Development Board
- Xilinx Platform USB Cable or Parallel IV Cable
- RS232 Cable
- A crossover ethernet cable connecting the board to a Windows or Linux host.
- Serial Communications Utility Program, such as HyperTerminal or Teraterm
- Xilinx Platform Studio 11.1i and ISE® 11.1 for making hardware modifications
- Xilinx SDK 11.1 for running or making modifications to the software

Introduction

lwIP is an open source networking stack designed for embedded systems. It is provided under a BSD style license. The objective of this application note is to describe how to use lwIP shipped along with the Xilinx EDK to add networking capability to an embedded system. In particular, this application note describes how applications such as an echo server or a web server can be written using lwIP.

Reference System Specifics

The reference design for this application note is structured in the following way. The ml505, ml507 and s3an folders correspond to the 3 supported boards. In each of these folders, the `hw` subdirectory contains the XPS hardware design, and the `sw` subdirectory contains the application software and software platforms that need to be imported into SDK.

The `memfs` folder contains the contents of the memory file system (MFS) image. The image itself is also present as the `image.mfs` file.

A copy of the old 10.1 based design is provided in the `10.1` folder. This folder is provided for reference purposes only, and is not described further in this application note.

Hardware Systems

The hardware systems for the three boards were built using Base System Builder (BSB), with minor modifications in XPS. To get more information about hardware requirements for lwIP, see the lwIP documentation available as part of EDK installation. Table 1 provides a summary of the hardware designs for the three boards:

Table 1: **Hardware Design Details**

Board	Processor	Processor Frequency	EMAC	DMA
ML505	MicroBlaze	125 MHz	xps_ll_temac	SDMA
ML507	PowerPC 440	400 MHz	xps_ll_temac	HDMA
S3-AN	MicroBlaze	62.5 MHz	xps_ethernetlite	None

Software Applications

The reference design includes five software applications:

1. Echo server
2. Web server
3. TFTP server
4. TCP RX throughput test
5. TCP TX throughput test

All of these applications are available in both RAW and Socket modes.

Echo Server

The echo server is a simple program that echoes whatever input is sent to the program via the network. This application provides a good starting point for investigating how to write lwIP applications.

The socket mode echo server is structured as follows. A main thread listens continually on a specified echo server port. For each connection request, it spawns a separate echo service thread, and then continues listening on the echo port.

```
while (1) {
    new_sd = lwip_accept(sock, (struct sockaddr *)&remote, &size);
    sys_thread_new(process_echo_request, (void*)new_sd,
DEFAULT_THREAD_PRIO);
}
```

The echo service thread receives a new socket descriptor as its input on which it can read received data. This thread does the actual echoing of the input to the originator.

```
while (1) {
    /* read a max of RECV_BUF_SIZE bytes from socket */
    n = lwip_read(sd, recv_buf, RECV_BUF_SIZE));

    /* handle request */
    nwrote = lwip_write(sd, recv_buf, n));
}
```

Note: The above code snippets are not complete and are intended to show the major structure of the code only.

The socket mode provides a simple API that blocks on socket reads and writes until they are complete. However, the socket API requires many pieces to achieve this, chief among them being a simple multithreaded kernel (xilkernel). Because this API contains significant overhead for all operations, it is slow.

The RAW API provides a callback style interface to the application. Applications using the RAW API register callback functions to be called on significant events like accept, read or write. A RAW API based echo server is single threaded, and all the work is done in the callback functions. The main application loop is structured as follows.

```
while (1) {
    xemacif_input(netif);
    transfer_data();
}
```

The function of the application loop is to receive packets constantly (`xemacif_input`), then pass them on to lwIP. Before entering this loop, the echo server sets up certain callbacks:

```
/* create new TCP PCB structure */
pcb = tcp_new();

/* bind to specified @port */
err = tcp_bind(pcb, IP_ADDR_ANY, port);

/* we do not need any arguments to callback functions */
tcp_arg(pcb, NULL);

/* listen for connections */
pcb = tcp_listen(pcb);

/* specify callback to use for incoming connections */
tcp_accept(pcb, accept_callback);
```

This sequence of calls creates a TCP connection and sets up a callback on a connection being accepted. When a connection request is accepted, the function `accept_callback` is called asynchronously. Because an echo server needs to respond only when data is received, the accept callback function sets up the receive callback by performing:

```
/* set the receive callback for this connection */
tcp_recv(newpcb, recv_callback);
```

When a packet is received, the function `recv_callback` is called. The function then echoes the data it receives back to the sender:

```
/* indicate that the packet has been received */
tcp_recved(tpcb, p->len);

/* echo back the payload */
err = tcp_write(tpcb, p->payload, p->len, 1);
```

Although the RAW API is more complex than the SOCKET API, it provides much higher throughput because it does not have a high overhead.

Web Server

A simple web server implementation is provided as a reference for a TCP based application. The web server implements only a subset of the HTTP 1.1 protocol. Such a web server can be used to control or monitor an embedded platform via a browser. The web server demonstrates the following three features:

- Accessing files residing on a Memory File System via HTTP GET commands
- Controlling the LED lights on the development board using the HTTP POST command
- Obtaining status of DIP switches (push buttons on the ML403) on the development board using the HTTP POST command

The Xilinx Memory File System (xilmfs) is used to store a set of files in the memory of the development board. These files can then be accessed via a HTTP GET command by pointing a web browser to the IP address of the development board and requesting specific files.

Controlling or monitoring the status of components in the board is done by issuing POST commands to a set of URLs that map to devices. When the web server receives a POST command to a URL that it recognizes, it calls a specific function to do the work that has been requested. The output of this function is sent back to the web browser in Javascript Object Notation (JSON) format. The web browser then interprets the data received and updates its display.

The overall structure of the web server is similar to the echo server – there is one main thread which listens on the HTTP port (80) for incoming connections. For every incoming connection, a new thread is spawned which processes the request on that connection.

The http thread first reads the request, identifies if it is a GET or a POST operation, then performs the appropriate operation. For a GET request, the thread looks for a specific file in the memory file system. If this file is present, it is returned to the web browser initiating the request. If it is not available, a HTTP 404 error code is sent back to the browser.

In socket mode, the http thread is structured as follows:

```
/* read in the request */
if ((read_len = read(sd, recv_buf, RECV_BUF_SIZE)) < 0)
    return;

/* respond to request */
generate_response(sd, recv_buf, read_len);
```

Pseudo code for the generate response function is shown below:

```
/* generate and write out an appropriate response for the http request */
int generate_response(int sd, char *http_req, int http_req_len)
{
    enum http_req_type request_type =
                         decode_http_request(http_req, http_req_len);

    switch(request_type) {
    case HTTP_GET:
        return do_http_get(sd, http_req, http_req_len);
    case HTTP_POST:
        return do_http_post(sd, http_req, http_req_len);
    default:
        return do_404(sd, http_req, http_req_len);
    }
}
```

The RAW mode web server does most of its work in callback functions. When a new connection is accepted, the accept callback function sets up the send and receive callback functions. These are called when sent data has been acknowledged or when data is received.

```
err_t accept_callback(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    /* keep a count of connection # */
    tcp_arg(newpcb, (void*)palloc_arg());

    tcp_recv(newpcb, recv_callback);
    tcp_sent(newpcb, sent_callback);

    return ERR_OK;
}
```

When a web page is requested, the `recv_callback` function is called. This function then performs work similar to the socket mode function – decoding the request and sending the appropriate response.

```
/* acknowledge that we've read the payload */
tcp_recved(tpcb, p->len);

/* read and decipher the request */
/* this function takes care of generating a request, sending it,
* and closing the connection if all data has been sent. If
* not, then it sets up the appropriate arguments to the sent
* callback handler.
*/
generate_response(tpcb, p->payload, p->len);

/* free received packet */
pbuf_free(p);
```

The complexity lies in how the data is transmitted. In the socket mode, the application sends data using the lwip_write API. This function blocks if the TCP send buffers are full. However in RAW mode, the application assumes the responsibility of checking how much data can be sent and of sending that much data only. Further data can be sent only when space is available in the send buffers. Space becomes available when sent data is acknowledged by the receiver (the client computer). When this event happens, lwIP calls the `sent_callback` function, indicating that data was sent which implies that there is now space in the send buffers for more data. The `sent_callback` is hence structured as follows.

```
err_t sent_callback(void *arg, struct tcp_pcb *tpcb, u16_t len)
{
    int BUFSIZE = 1024, sndbuf, n;
    char buf[BUFSIZE];
    http_arg *a = (http_arg*)arg;

    /* if connection is closed, or there is no data to send */
    if (tpcb->state > ESTABLISHED) {
        return ERR_OK;
    }

    /* read more data out of the file and send it */
    sndbuf = tcp_sndbuf(tpcb);
    if (sndbuf < BUFSIZE)
        return ERR_OK;

    n = mfs_file_read(a->fd, buf, BUFSIZE);
    tcp_write(tpcb, buf, n, 1);

    /* update data structure indicating how many bytes
     * are left to be sent
     */
    a->fsize -= n;
    if (a->fsize == 0) {
        mfs_file_close(a->fd);
        a->fd = 0;
    }

    return ERR_OK;
}
```

Both the sent and the receive callbacks are called with an argument which can be set using **`tcp_arg`**. For the web server, this argument points to a data structure which maintains a count of how many bytes remain to be sent and what is the file descriptor that can be used to read this file.

TFTP Server

TFTP (Trivial File Transfer Protocol) is a UDP based protocol for sending and receiving files. Because UDP does not guarantee reliable delivery of packets, TFTP implements a protocol to ensure packets are not lost during transfer. Detailed explanation of the TFTP protocol can be found in the *RFC 1350 – The TFTP Protocol*.

The socket mode TFTP server is very similar to the web server in application structure. A main thread listens on the TFTP port and spawns a new TFTP thread for each incoming connection request. This TFTP thread implements a subset of the TFTP protocol and supports either read or write requests. At the most, only one TFTP Data or Acknowledge packet can be in flight which greatly simplifies the implementation of the TFTP protocol. Because the RAW mode TFTP server is very simplistic and does not handle timeouts, it is usable only as a point to point ethernet link with zero packet loss. It is provided as a demonstration only.

Because TFTP code is very similar to the web server code explained above, is not explained in this application note. Because of the use of UDP, the minor differences can be easily understood by looking at the source code.

TCP RX Throughput Test and TCP TX Throughput Test

The TCP transmit and receive throughput test applications are very simple applications that have been written to determine the maximum TCP transmit and receive throughputs achievable using lwIP and the Xilinx EMAC adapters. Both these tests communicate with an open source software called iperf (http://iperf.sourceforge.net/).

The transmit test measures the transmission throughput from the board running lwIP to the host. In this test, the lwIP application connects to an iperf server running on a host, and then keeps sending a constant piece of data to the host. Iperf running on the host determines the rate at which data is being transmitted and prints that out on the host terminal.

The receive test measures the maximum receive transmission throughput of data at the board. The lwIP application acts as a server. This server can accept connections from any host at a certain port. It keeps receiving the data that is sent to it, and just silently drops all that received data. iperf (client mode) on the host can connect to this server and transmit data to it for as long as needed. At frequent intervals, it computes how much data has been transmitted at what throughput and prints this out on the console.

Creating an lwIP Application Using the Socket API

The software applications provide a good starting point to write other applications using lwIP. lwIP socket API is very similar to the Berkeley/BSD sockets, therefore, there should be no issues writing the application itself. The only difference lies in the initialization process which is coupled to lwip130 library and xilkernel.

The three sample applications all utilize a common main.c file for initialization and to start processing threads. Perform the following steps for any socket mode application.

1. Configure the xilkernel with a static thread. In the sample applications, this thread is given the name `main_thread`. In addition, make sure xilkernel is properly configured by specifying the system interrupt controller. lwIP also requires yield functionality available in xilkernel only when the 'enhanced_features' parameter of xilkernel is turned on.
2. The main thread initializes lwip using the `lwip_init` function call, and then launches the network thread using the `sys_thread_new` function. Note that all threads that use the lwIP socket API must be launched with the `sys_thread_new` function provided by lwIP.
3. The main thread adds a network interface using the `xemac_add` helper function. This function takes in the IP address and the MAC address for the interface, and initializes it.
4. The `xemacif_input_thread` is then started by the network thread. This thread is required for lwIP operation when using the Xilinx adapters. This thread takes care of

moving data received from the interrupt handlers to the `tcpip_thread` that is used by lwIP for TCP/IP processing.

5. The lwIP library has now been completely initialized and further threads can be started as the application requires.

Creating an lwIP application using the RAW API

The lwIP RAW mode API is more complicated to use as it requires knowledge of lwIP internals. The typical structure of a RAW mode program is as follows.

1. The first step is to initialize all lwIP structures using **`lwip_init`**.
2. Once lwIP has been initialized, an EMAC can be added using the `xemac_add` helper function.
3. Because the Xilinx lwIP adapters are interrupt based, enable interrupts in the processor and in the interrupt controller.
4. Set up a timer to interrupt at a constant interval. Usually, the interval is around 250 ms. Update the tcp timers at every timer interrupt.
5. Once the application is initialized, the main program enters an infinite loop performing packet receive operation, and any other application specific operation it needs to do.
6. The packet receive operation (`xemacif_input`), processes packets received by the interrupt handler, and passes them onto lwIP, which then calls the appropriate callback handlers for each received packet.

Executing the Reference System

This section describes how to execute the reference design and the expected results.

Note: This section provides details specifically for the ML505 design. The steps are the same for the other two designs, except for the address at which the memory file system (MFS) is loaded. The correct address to be used for loading the MFS image should be determined by looking at the corresponding software platform settings for xilmfs library.

Host Network Settings

Connect the FPGA board to an Ethernet port on the host computer via a cross-over cable. Assign an IP address to the Ethernet interface on the host computer. The address must be the same subnet as the IP address assigned to the board. The software application assigns a default IP address of 192.168.1.10 to the board. The address can be changed in the `apps/src/main.c` file. For this setting, assign an IP address to the host in the same subnet mask, for example `192.168.1.100`.

Compiling and Running the Software

The reference applications can be compiled and run using the Xilinx Software Development Toolkit (SDK) with the following steps (explained in further detail in the subsequent paragraphs):

1. Open up SDK in a new workspace and provide a reference to the hardware platform xapp1026/ml505/hw/SDK/SDK_Export/hw/system.xml.
2. Import the software platform and software applications. This will automatically compile both the software platform and the applications.
3. Download the bitstream
4. Download the MFS image.
5. Create a run configuration and run the application.

Follow the same steps to import and run any application using SDK. For more details regarding SDK concepts and tasks, please refer to the online help in SDK.

Step 1: Specify the Workspace and the Hardware Platform

Eclipse organizes projects within a folder called workspace. In SDK, a workspace can only contain projects for one specific hardware platform. So when SDK starts up, please specify a folder that will contain software projects for a particular hardware design.

The next step is to provide a reference to the hardware platform for which the software is being developed. This is done by pointing SDK to the hardware design specification file exported from XPS. In the reference design, this specification is available in the `xapp1026/ml505/hw/SDK/SDK_Export/hw/` folder.

Step 2: Import Software Projects

Software platforms and applications can now be created in SDK once the hardware platform has been specified. Rather than creating a new software platform/application, you can just import the existing software platforms and example applications provided with this reference design. This can be achieved via the following steps:

1. Click on the File->Import menu item. This opens up an import wizard.
2. Select Existing SDK Projects into Workspace in the import wizard.
3. To select the root directory from which the projects need to be imported, click on Browse, and specify the location where the software applications have been stored. For the ml505 design, this location is xapp1026/ml505/sw folder.

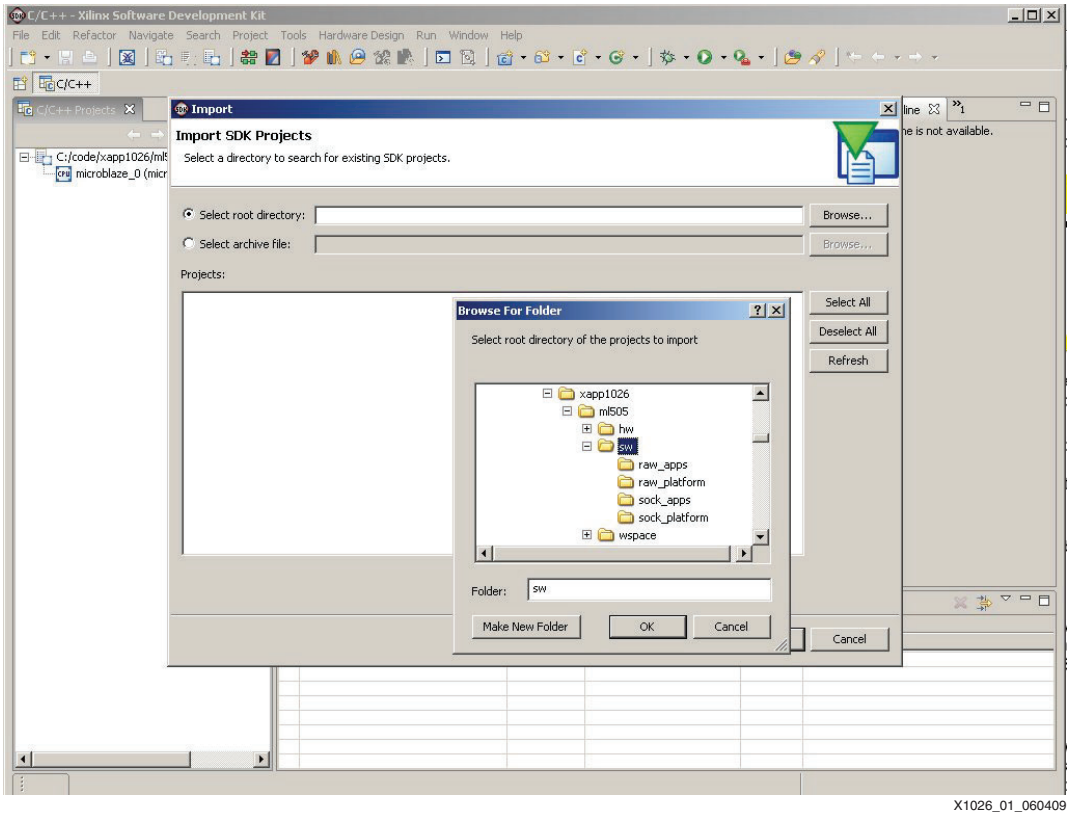


Figure 1: **Select Folder to Import Sources**

4. The import wizard should show a list of projects that are available to import. This list should show the following 4 projects: raw_apps, raw_platform, sock_apps, and sock_platform. Select all 4 projects to be imported, and click **Finish**.

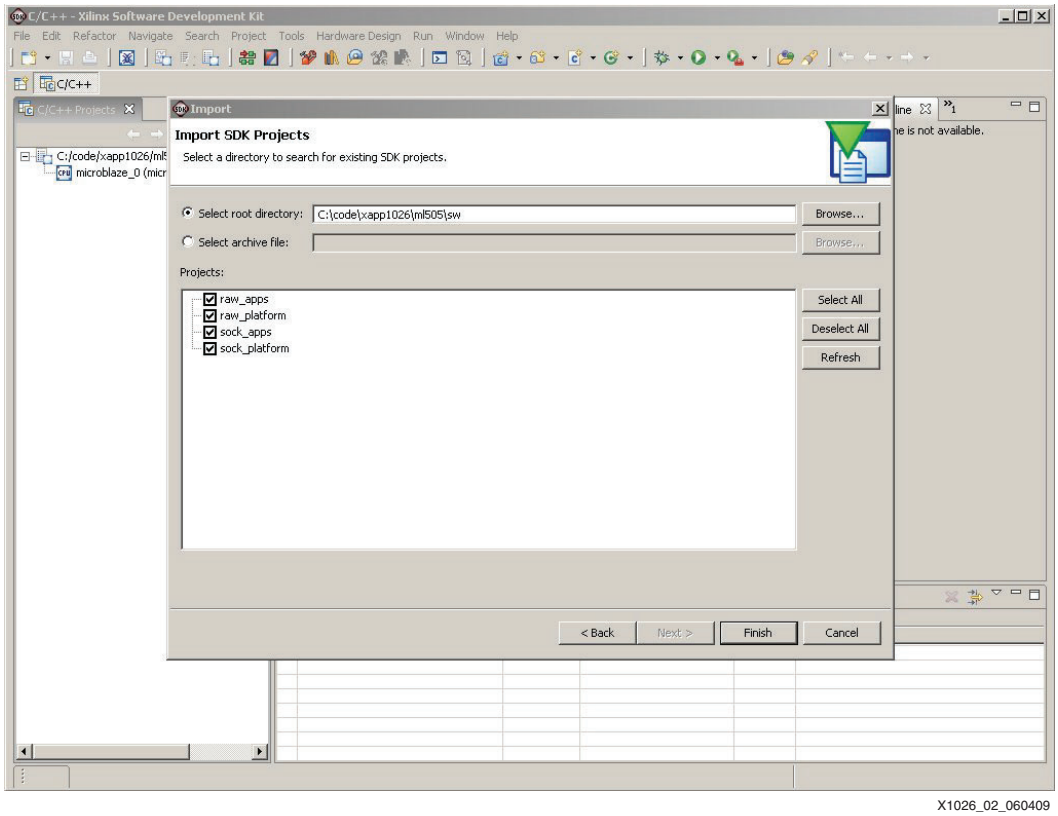


Figure 2: **Select Projects to Import**

Click **Yes** every time that SDK prompts about overwriting an existing `.project` file.

Note: The last step takes a significant amount of time, because SDK compiles the software platform and the software applications in the background. However, during this time the GUI will be unresponsive. This issue has been fixed in 11.1 Update 1.

Step 3: Download the Bitstream

To download the bitstream, select the Tools ' Program FPGA menu item. Make sure that the bitstream points to the correct bitstream for the board. For ML505, confirm that the bitsream points to xapp1026/ml505/hw/SDK/SDK_Export/hw/system.bit, then click **Save and Program**.

Step 4: Download the MFS Image

The memory file system image contains the files required for the web server to serve files from, and for the TFTP server to store or retrieve files. This image has to be downloaded to the on board external memory before the executable can run properly. To download the MFS image, first click on Tools ' XMD console. From within XMD, navigate to the location where the reference design has been unzipped by issuing a cd command (e.g: cd /code/xapp1026/memfs). From this location, download the image by issuing the following command:

```
XMD% connect mb mdm (for ml507 design, use "connect ppc hw")
XMD% dow -data image.mfs 0x51000000
```

Note that the address specified above is valid only for the ML505 design. For the ML507 design, download the image to address `0x01f00000`, and for the S3-AN design, download it to address `0x47000000`. These addresses are specified in the xilmfs library configuration that

can be viewed in the software platform settings dialog. See Appendix A for instruction on how to create the MFS image.

Step 5: Create a run configuration and run the application

To run the application, first create a run configuration specifying the ELF that needs to be run. To create a Run configuration, select Run ' Run. Then click on New to create a new run configuration. Specify the ELF that needs to be run: use raw_apps.elf to run the RAW mode example, and sock_apps.elf to run the socket mode example. Finally, click on Run to actually run the executable.

Interacting with the Running Software

Both the socket mode and the raw mode applications bundle the following examples together into a single executable: echo server, web server, TFTP server, receive and transmit throughput tests. Once the executable has been run, the following output should appear on the serial port.

Output from the Application

```
-----lwIP RAW Mode Demo Application ------
Board IP:       192.168.1.10
Netmask :       255.255.255.0
Gateway :       192.168.1.1
auto-negotiated link speed: 1000

              Server   Port Connect With..
-------------------- ------ --------------------
         echo server      7 $ telnet <board_ip> 7
       rxperf server   5001 $ iperf -c <board ip> -i 5 -t 100
       txperf client    N/A $ iperf -s -i 5 -t 100 (on host with IP
192.168.1.100)
         tftp server     69 $ tftp -i 192.168.1.10 PUT <source-file>
         http server     80 Point your web browser to http://192.168.1.10
```

For the socket mode application, only the first line changes to indicate that it is the socket mode demo application. At this point, you can interact with the application running on the board from the host machine.

Interacting with the Echo Server

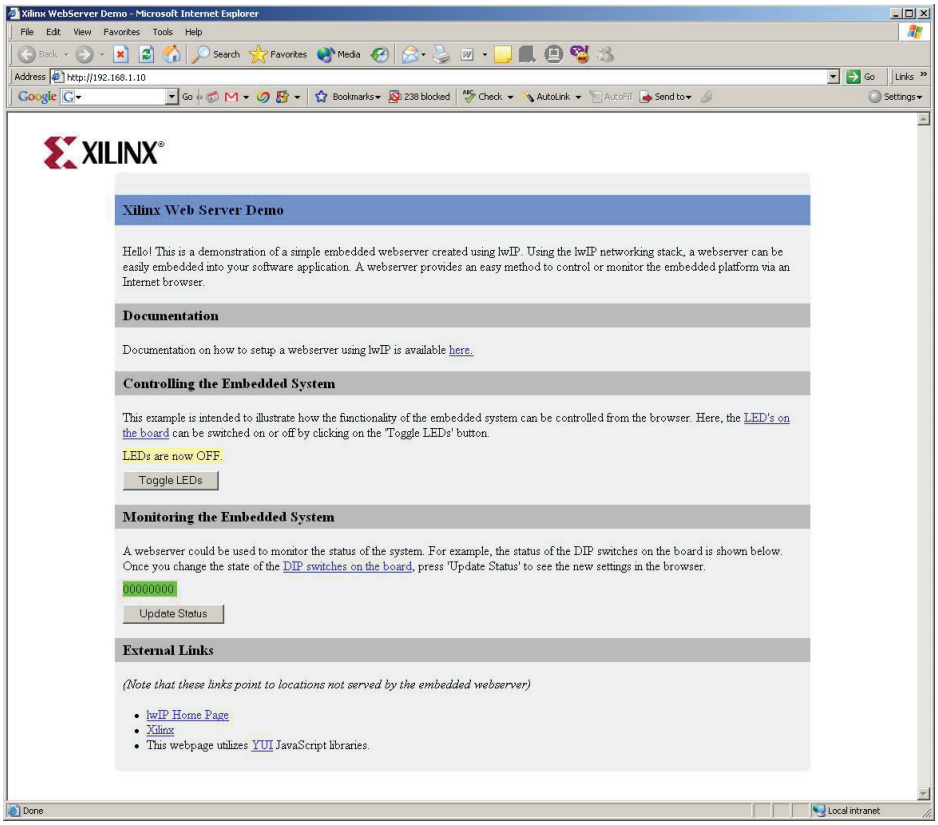
To connect to the echo server, use the telnet utility program.

```
$ telnet 192.168.1.10 7
Trying 192.168.1.10...
Connected to 192.168.1.10.
Escape character is '^]'.
hello
hello
world world ^]
telnet> quit
Connection closed.
```

If the echo server works properly, any data sent to the board will be echoed in response as seen above. Note that certain telnet clients will immediately send the character to the server and echo the received data back rather than waiting for the carriage return.

Interacting with the Web Server

Once the web server is active, it can be connected to using a web browser. The sample web pages use Javascript, so the browser must have javascript enabled. A sample screen shot of the web page that is served is shown in Figure 3. The Toggle LED button will toggle the state of the LEDs on the board. Clicking the Update Status button will refresh the status of the DIP switches on the web page. These are intended to show simple control and monitoring of the embedded platform via the ubiquitous web browser. Note that the external links section contains links that point to content that are not served by the development platform.



X1026_03_060409

Figure 3: **Web Page Served by the Reference Web Server**

Interacting with the TFTP Server

The TFTP server provides simple file transfer capability to and from the memory file system resident on the board. An example interaction is shown below using the TFTP client on Windows.

C:\>tftp -i 192.168.1.10 GET index.html

Transfer successful: 2914 bytes in 1 second, 2914 bytes/s

C:\>tftp -i 192.168.1.10 PUT test.txt

Transfer successful: 19 bytes in 1 second, 19 bytes/s

The above two examples show how to read or write files on the board's memory file system from the local host.

Interacting with the Receive Throughput Test

To measure receive throughput, connect to the receive iperf application using the iperf client.

```
$ iperf -c 192.168.1.10 -i 5 -t 50 -w 8k
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 16.0 KByte (WARNING: requested 8.00 KByte)
------------------------------------------------------------
[  3] local 192.168.1.100 port 35928 connected with 192.168.1.10 port 5001
[  3]  0.0- 5.0 sec  91.4 MBytes    153 Mbits/sec
[  3]  5.0-10.0 sec  91.1 MBytes    153 Mbits/sec
[  3] 10.0-15.0 sec  91.6 MBytes    154 Mbits/sec
[  3] 15.0-20.0 sec  91.5 MBytes    153 Mbits/sec
[  3] 20.0-25.0 sec  91.6 MBytes    154 Mbits/sec
[  3] 25.0-30.0 sec  92.3 MBytes    155 Mbits/sec
[  3] 30.0-35.0 sec  92.3 MBytes    155 Mbits/sec
[  3] 35.0-40.0 sec  92.3 MBytes    155 Mbits/sec
[  3] 40.0-45.0 sec  92.3 MBytes    155 Mbits/sec
[  3]  0.0-50.0 sec    919 MBytes    154 Mbits/sec
```

Note: To achieve maximum throughput numbers, ensure that the executable has been compiled for Release (-O2 optimization level) rather than for Debug (-O0 optimization).

Interacting with the Transmit Throughput Test

To measure the transmit throughput, first start the iperf server on the host, then run the executable on the board. When the executable is run, it tries to connect to a server at host 192.168.1.100. This address can be changed in the file txperf.c. A sample session is as follows:

```
$ iperf -s -i 5
------------------------------------------------------------
Server listening on TCP port 5001TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.1.100 port 5001 connected with 192.168.1.10 port 4097
[  4]  0.0- 5.0 sec  78.3 MBytes    131 Mbits/sec
[  4]  5.0-10.0 sec  78.9 MBytes    132 Mbits/sec
[  4] 10.0-15.0 sec  79.3 MBytes    133 Mbits/sec
[  4] 15.0-20.0 sec  79.2 MBytes    133 Mbits/sec
[  4] 20.0-25.0 sec  79.2 MBytes    133 Mbits/sec
[  4] 25.0-30.0 sec  79.2 MBytes    133 Mbits/sec
[  4] 30.0-35.0 sec  79.2 MBytes    133 Mbits/sec
[  4] 35.0-40.0 sec  79.2 MBytes    133 Mbits/sec
[  4] 40.0-45.0 sec  79.2 MBytes    133 Mbits/sec
[  4] 45.0-50.0 sec  79.1 MBytes    133 Mbits/sec
[  4] 50.0-55.0 sec  79.2 MBytes    133 Mbits/sec
^C Waiting for server threads to complete. Interrupt again to force quit.
^C
```

Press Ctrl + C twice to stop the server.

lwIP Performance

The receive and transmit throughput applications are used to measure the maximum TCP throughput possible with lwIP using the Xilinx ethernet adapters. The following table summarizes the results for different configurations.

Table 2: **TCP Receive and Throughput Results**

Design	RAW Mode		Socket Mode	
	RX	TX	RX	TX
ml505	150 Mbps	130 Mbps	25 Mbps	39 Mbps

Table 2: **TCP Receive and Throughput Results**

Design	RAW Mode		Socket Mode	
	RX	TX	RX	TX
ml507	500 Mbps	440 Mbps	160 Mbps	240 Mbps
s3an	25 Mbps	15 Mbps	8 Mbps	9 Mbps

These performance numbers were obtained under the following conditions:

1. The hardware designs used are the same ones present in the reference designs.
2. When measuring receive throughput, only the receive throughput application was enabled (in file config_apps.h). Similarly, only the transmit throughput test thread was present in the ELF when measuring the transmit throughput.
3. The host machine was a Dell desktop running Fedora 8. The NIC card used on the host was based on Intel Corporation 82572EI Gigabit Ethernet Controller.

For information on how to optimize the host setup, and benchmarking TCP in general, please consult XAPP 1043.

Note: This application note may not be updated for every EDK release. For the latest TCP throughput results, please consult the lwIP documentation provided as part of EDK.

Debugging Network Issues

If any of the sample applications do not work, there could be a number of potential reasons. This section provides a troubleshooting guide to fix common sources of setup errors.

1. First, ensure that the link lights are active. Most development boards have LEDs that indicate whether an ethernet link is active. If the bitstream downloaded has some ethernet MAC properly configured, and a ethernet cable is attached to the board, the link lights should indicate an established ethernet link.
2. If the board includes LEDs indicating the link speed (10/100/1000 Mbps), verify that the link is established at the correct speed. For designs that include xps_ethernetlite EMAC IP, the link should be established at only 10 or 100 Mbps. xps_ethernetlite will not be able to transmit or receive data at 1000 Mbps. The xps_ll_temac EMAC core supports all three link speeds. The TEMAC must be informed of the correct speed to which the PHY has auto-negotiated. lwIP includes software to detect the PHY speed, however this software works only for Marvell PHYs. Users should confirm that the link speed that lwIP detects matches the link speed as shown in the LEDs.
3. To confirm that the board actually receives packets, the simplest test is to ping the board, and check to make sure that the RX LED goes high for a moment to indicate that the PHY actually received the packet. If the LEDs do not go high, then there are either ARP, IP, or link level issues that prevent the host from sending packets to the board.
4. Assuming that the board receives the packets, but the system does not respond to ping requests, the next step is to ensure that lwIP actually receives these packets. This can be determined by setting breakpoints at XEmacLite_InterruptHandler for xps_ethernetlite systems, and lldma_recv_handler for xps_ll_temac systems. If packets are received properly, then these breakpoints should be hit for every received packet. If these breakpoints are not hit, then that indicates that the MAC is not receiving the packets. This could mean that the packets are being dropped at the PHY. The most common reason that the breakpoints are not hit, is that the link was established at a speed that the EMAC does not support.
5. Finally, some hosts have firewalls enabled that could prevent receiving packets back from the network. If the link LEDs indicate that the board is receiving and transmitting packets, yet the packets transmitted by the board are not received in the host, then the host firewall settings should be relaxed.

When these applications are ported over to a different board or hardware, care should be taken to make sure that there is sufficient heap and stack space available (as specified in the linker script).

Conclusion

This application note showcases how lwIP can be used to develop networked applications for embedded systems on Xilinx FPGAs. The echo server provides a simple starting point for networking applications. The web server application shows a more complex TCP based application, and the TFTP server shows a complex UDP based application. Applications to measure receive and transmit throughput provide an indication of the maximum possible throughput using lwIP with Xilinx adapters.

Creating an MFS Image

To create an MFS image from the contents of a folder, use the following command from a EDK bash shell. From SDK, do Tools ' Launch Shell to get to the bash shell.

```
cd memfs
memfs$ mfsgen -cvbfs ../image.mfs 1500 *
mfsgen
Xilinx EDK 11.1 EDK_L.29.1
Copyright (c) 2004 Xilinx, Inc.  All rights reserved.

cmd  34
css:
main.css  744
images:
board.jpg  44176
favicon.ico  2837
logo.gif  1148
index.html  2966
js:
main.js  7336
yui:
anim.js  12580
conn.js  11633
dom.js  10855
event.js  14309
yahoo.js  5354
MFS block usage (used / free / total) = 234 / 1266 / 1500
Size of memory is 798000 bytes
Block size is 532
mfsgen done!
```

References

1. lwIP – A Lightweight TCP/IP Stack– CVS Repositories
2. *RFC 1350 – The TFTP Protocol*
3. iperf software
4. XAPP1043 *Measuring Treck TCP/IP Performance Using the XPS LocalLink TEMAC in an Embedded Processor*

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
10/13/08	1.0	Initial Xilinx release.
6/15/09	2.0	Updated to v2.0 for IDS11.1.

Notice of Disclaimer